Chapter 17

INTERNATIONALISATION TECHNIQUES IN DEVELOPMENT OF MULTILINGUAL SOFTWARE

Agenor Hofmann-Delbor

1. THE ROLE OF THE PRODUCTION STAGE WITH REGARD TO MULTILINGUAL SOFTWARE

Back in the old days there were two fully separate categories of products: those oriented for the local market, and those aimed at the global market. In this age of globalisation, short supply chains, quick product delivery and constant optimisation of production processes, it is not difficult to say that most products have become global. The products that cannot be transformed in order to be sold in different countries or regions are limited mainly to those based on culture or religion. This also means that virtually any product can be prepared and customised for global end users. Only three conditions need to be met:

1. Products need to be translated for local languages
2. Products need to work properly with all planned functionalities
3. Products need to be delivered to local customers

Before going into detail, let us define the terminology used in this chapter. There are three main terms used:

Globalisation – for the purpose of this chapter let us define globalisation as all the processes that allow reaching customers or end users in different countries all around the world, and all the processes that allow manufacturing of products in various time zones and regions. This is also referred to by the *g18n* acronym.

Localisation – translation of content in a way that allows meeting the technical and cultural standards of a specific region or country. This is often referred to as translation+, showing that it creates a product more adapted to the needs of a local market, not just speaking the language of the end user. There are many technical

requirements for proper localisation as we will describe later. It is also referred to by the *l18n* acronym.

Internationalisation – the process of creating products which can be localised into several languages. It should always precede the localisation stage. This process does not assume the author knows other cultural languages. However, it creates a set of rules and good practices that should be followed if further localisation is planned. Also referred to by the *i10n* acronym.

It is worth mentioning that many sources use the term "globalisation" as a synonym for internationalisation when it comes to software development. For the purpose of this chapter we will separate these terms.

2. TIME-TO-MARKET AND SOFTWARE DEVELOPMENT

Time-to-market has been the leading marketing and sales strategy used successfully for years. There are many techniques involved in this process but the most important goal is to simply win the largest possible part of a specific market. Again, there are dozens of ways to do it, but the most important two are:

- Creating a product as quickly as possible to reach the market before the competition,
- Delivering the product simultaneously to several markets.
- The globalisation process has made this approach a common technique used by software manufacturers.

However, before the time-to-market strategy comes in place, the developers need to focus on the product. Before we even think about delivery, it is critical to have "deep understanding of the customer and how the product is likely to be used" (Kahn, 2004). In terms of software development, this may be a difficult task as the product is usually completed in the very last stage of production. This is where internationalisation comes into place with the many concepts that in practice prepare the product for localisation, and also – for its global presence.

The time-to-market strategy, regardless of its popularity, can also lead to bottleneck issues. One recent example was the very loud release of the video game Diablo III by Blizzard Entertainment. Although this is just an entertainment product, video games are a serious, rapidly growing part of the software development industry and follow the same localisation process. The release of this product showed a classic example of how a time-to-market strategy may sometimes lead to huge problems. Blizzard released the product simultaneously in more than 30 countries worldwide and also on two platforms: PC and Mac. Due to it's outstanding demand and popularity, stocks were quickly cleared. This may have been a commercial success, and obviously was, but this success also had a downside. Since the release was, to a degree, synchronised worldwide to enhance the marketing effect, all the end users started using the product at the same time.

Due to it's licensing policy, to run the program each user needed to login to Blizzard's servers. The outcome was similar to a DDOS attack on a large scale (http://www.diablo3community.com/content.php/19-Diablo-3-Release-Date-Diablo-iii-Release-Date). It took a few days to solve the problems with connections. This relatively harmless example shows that the time-to-market strategy does not end on the release date. It should be closely assigned to the product's life cycle.

It is also important to mention that the vast amount of the software localisation projects are not generated by typical software manufacturers. Such projects are common in other industries, such as in pharmaceuticals, or automotive. Virtually all electronic devices use some variation of display screens and a user interface, most of them also include separate documentation, which is usually considered to be part of the software localisation project.

3. MULTILINGUAL SOFTWARE: PROJECTS AND COSTS

One of the truisms we must use here is that all software projects are all about numbers – costs, sizes, dates. Any decision made at the preparation stage may and usually do have a significant impact on the final budget.

To understand the impact of all action taken by software developers, we should first estimate the size of the market. Software localisation is part of the outsourced language service. The outsourced language services market in total is estimated to be worth $33.523 billion in 2012 (Kelly, DePalma & Stewart, 2012). Unfortunately the majority of localisation projects are confidential, so it is almost impossible to say if software localisation is 1%, 5% or 10% of the market in total. However, we have some clues. Without any doubt we know that there are a growing number of applications, platforms and devices available. A typical large localisation project is usually worth a couple of million dollars, and just by looking at translation jobs published officially (ProZ, TranslatorsCafe etc.) we can say that there are thou-sands of localisation projects out there, including video games (each title is a software localisation project), devices (mobile phones, applications) and other products. Just by looking at these assumptions we may expect the software localisation market to be worth roughly 1 billion US$ (1000 projects * 1 million). If we need to look at localisation projects globally, then so called Multi-Language Vendors (companies, which deliver translations into several languages) handle such jobs. If a single language is worth 100,000 EUR (which means the size of the project is more or less 800,000 words – rather large, yet still an average software project), multiplying that by the number of languages quickly brings us over six digits. This shows how underestimated this (1 billion dollar) assumption can be.

4. SOFTWARE LOCALISATION: ERRORS AND TESTING

Localisation errors are typically handled and treated as less critical than functional errors. However, there are many cases where this could be a false assumption. In general, each serious error converts into a loss of money. Just by looking at the United Stated we need to notice that "*software defects cost the U.S. economy almost $60 billion annually*" (Vadher, 2010). And that's not even ¼ of the global market if we look at their economy.

The testing stage is quite natural for any software development project, albeit it may not be clear how much difference it makes if errors are avoided during the production stage instead of being fixed afterwards. Fortunately there is some reliable information on this matter. A study conducted by the Systems Sciences Institute at IBM has shown some interesting findings. It seems the cost of fixing an error found after product release was over six times higher than one uncovered during the implementation phase, and up to 100 times higher if the error was identified during the maintenance phase. The graph below shows the comparison.

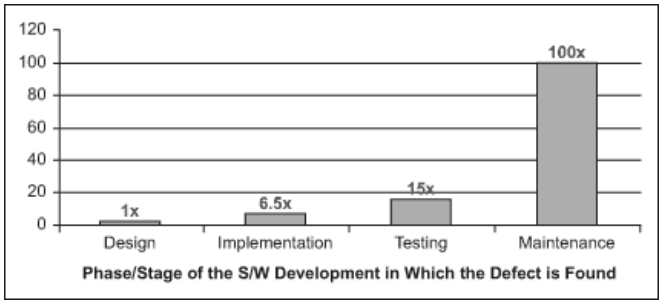

Fig. 17.1. Relative Costs to Fix Software Defects. Source: IBM Systems Sciences Institute

This shows how important it is to detect and prevent errors in the early stages – it simply allows outstanding savings. With the budgets described earlier, this gives manufactures a good reason to pay more attention to this subject. This also shows how important the testing stage is. The cost of fixing an error after software release is extremely high – for an error that could be fixed for $1000 during the design stage, we would need to pay up to $100,000 if detected during the maintenance phase. It needs to be said that some errors need to be fixed for reasons that go beyond just typical quality assurance, i.e. legal matters. This means either the company fixes the error or the product needs to be withdrawn from the market. If we now go back to the 100x multiplier of the initial costs, this could potentially be a threat to even large companies and their financial stability.

If we know how important testing and detecting errors is, let us discuss the efficiency of this process. A study based on average defect detection rates for 6 different JPL projects showed that on average testers found 1 defect every 4 hours

during black box testing (Kelly, Sherif, & Hops, 1992). This means if the software producer wants to identify just three errors he should allocate at least 12 working hours for testing. Obviously the tests usually last longer and have several iterations. This also shows how a limited number of testing hours reflects in the number of errors found. It needs to be said that localisation testing allows more efficient error detection rate as the texts are easier to display and check compared to testing functionalities and dependencies, yet according to the author's knowledge no public study has been conducted to confirm this in numbers. However, we cannot assume fixing localisation errors is any less time consuming for developers than fixing regular functional errors. It may be the case for minor localisation errors, but for some of the issues it may actually require more time than to fix other types of issues or even redesigning the core of the software. Some experiences show the "average time to find and fix each defect is generally 10 to 20 or more hours" (Vadher, 2010). We initially considered a project worth 100,000 EUR (800,000 words). Let us assume the project had been translated into 10 languages and for each language we only found 5 errors. Simple calculations show we need about 50 hours to fix errors for each language, assuming the issues only apply to a specific language. This makes 500 hours, plus work time required for testing – as we have previously estimated 4 hours per error, 20 hours multiplied by 10 (number of languages). The total shows a significant number of 700 hours. If the rate per hour is 30 EUR this makes almost 30,000 EUR, which is 30% of the initial budget required to translate the entire project. This example shows that software localisation is not an additional stage, it is one of the most crucial and financially demanding stages of the entire production process.

Obviously the time of the localisation tests would depend highly on the quality of both the code and the translating. Real life examples show many errors are actually caused by developers taking shortcuts and trying to "make life easier" for the translators and testers. This usually leads to multiple issues, as it is impossible for a programmer to know all the linguistic and cultural consequences of each decision related to extracting or displaying strings in a target language. We will expand on this topic in the following parts. Let us focus on one of the common issues – sorting software strings.

All texts used for UI localisation are part of the project resources. Resources are usually grouped in tables, assigning each string a unique ID. It doesn't make a difference if the table is stored in a text file (.resx, .properties etc.) or in an SQL database. The result is basically a list of numbered strings. In general - some of them are selected for translation, other remain locked. The simplest way to start the project is to select the strings, extract them (or allow extracting by specialised tools) and send them to the translation department or vendor. This being said, we may introduce a most surprising fact – for some reason one of the most common errors in project preparation is sorting strings alphabetically. It could make importing the strings back easier in some cases, but in most situations there is no logical reason for this. This issue affects the entire translation chain, especially the

translator, who sees the random strings from different parts of the software, dialog boxes etc. We should make it clear – translators and localisation testers live in a symbiosis. More translation errors naturally extend the time required for testing and make this phase more difficult. Real-life examples (unfortunately project names and manufacturers cannot be published) show some interesting conclusions – testing projects where the strings have initially been sorted alphabetically is 5 times more time consuming than for projects with strings grouped by subject. Reading between lines, we need to be aware that the difficult and long testing phase usually also means the linguistic quality of the projects is also poor.

With the popularity of software in general comes the popularity of data formats that are closely related. Even if the two main programming environments (JAVA and .NET) require a slightly different approach, both refer to common and universal data formats. Without any doubt, XML is the universal language for exchanging data, building information structures and markups. XML is a perfect example of how all stages of the localisation process influence the final cost of the project. Even documents as universal and flexible as XML can be built incorrectly and "inappropriate use of syntactical tools can have a profound effect on translatability and cost. It may even require complete re-authoring of documents in order to make them translatable" (Zydroń, 2004).

A very common technique used by developers is substituting variables with text. One of the ways this could be done in XML documents is by using entities. For example:

```
<translatable_text>Object &name cannot be be found.</translatable_text>
```

Unfortunately this results in numerous issues, because of the grammar inflection, word ordering, ordinal followers etc. This will be covered in following parts. Using entities can also cause problems with XML parsing and using translation memories (Zydroń, 2004). It is strongly suggested to use tags and IDs instead of entities. The previous example could be converted into:

```
<translatable_text>
Object <function id=″517″>calculate</function> cannot be found.
</translatable_text>
```

Authors of XML documents should also avoid using translatable attributes (eg. <function id="517" name="calculate">) and learn to properly use the CDATA section which usually contains escaped special characters and text, which should not be translated. It is quite common to see the translatable text embedded into this section, where this should not happen. Strings should not be split into several sentences. For example:

```
<para>
  <line>This text should not be</line>
  <line>broken this way – the translated
  text may well be in a different order.</line>
</para>
```

Example: Incorrect fragmentation of text. Source: Zydroń., 2004.

5. SOFTWARE LOCALISATION: DELAYS AND DEADLINES

Software projects are extremely vulnerable to delays and it is common knowledge that the linguistic part (i.e. translation and localisation) is not considered the key aspect of the process. There are currently two ways to localise the software:

1) translate on-the-fly with regard to AGILE programming. Each translation set is completed part by part, without actually having the running product. This enables a very rapid development, but makes the translation part more difficult, and could lead to lowering the quality of the localized product.
2) localise the product after the coding part has been completed. This approach gives the translator a chance to see the working products, but it requires a longer life cycle – therefore it is now a very rare case.

This shows that the localisation team is exposed to a huge pressure to meet deadlines and deliver the product as soon as possible. The main reason is, obviously, the increasing cost of the project. It all depends on the task – "the cost of delay may be less than $1000 per day, but for major projects in large companies in can exceed $1 million per day in pretax profit" (Kahn, 2004). One of the common situations in the localisation industry is keeping the 24/7 work cycle by locating development and translation teams in different time zones globally. This eventually reflects in the deadlines received by vendors. As we have shown delays could be really expensive and in many cases the main reason for them is not delivery to the end user, but rather internal costs, and synchronising tasks performed by staff.

6. INTERNATIONALISATION AND SOFTWARE DEVELOPMENT

The previous paragraphs have shown the role of localisation and the cost dependencies. We also focused on testing. As we said earlier, all the mentioned processes are a consequence of the decisions made in the early stage, referred to previously (Fig 17.1) as the designing stage. In fact this process is more complex and allows avoiding many errors and cutting down costs that would result from ignoring this process. We have shown the numbers and the importance of preventing errors in software production. We should now discuss the crucial process which combines all the experiences and findings – internationalisation.

There are two types of internationalisation: cultural and technical. Both of them may have different consequences, but ignoring either of them is simply dangerous. Let us briefly discuss the popular issues.

6.1. Internationalisation: Linguistic issues

One of the most common issues, and definitely the very first piece of information developers should learn about, is that English language has different grammar and linguistic rules than other languages. It is simply impossible to apply all the rules to other languages. However, it is always tempting, because English allows smooth automation of text flow, inserting entities into text (described in part 4) without affecting the text meaning and so on. In consequence many localisation projects cannot be properly localised into a specific language because it is not possible to keep the same grammar structure and word order in the target language. Developers tend to use placeholders to insert information generated by applications (in such cases there is always a risk of breaking the A6 internationalisation rule described in part 8). This may be a variable name, text, number, value or other type of information. Usually the created source text looks like the following sample:

```
The $variable_name of $other_variable is $mode_name.
```

The following example is extremely difficult to translate into Slavic languages with complicated inflection, grammar forms, ordinal followers etc. Even if the translator succeeds there is still the risk of finding the second most common issue – string concatenation (violation of rule A7 described in the part 8). Even though it is forbidden by all large manufacturers, translators still report cases like this. Let us look at the example below:

```
$String_2354=John
$String_2355=feeds
$String_2356=his
$String_2357=cat.
```

Obviously the developer expect a simple operation of concatenating those strings (`$String_2354+whitespace+$String_2355+whitespace+$String_2356 +whitespace+$String_2357`) to result in a perfectly correct sentence. Unfortunately in most languages this only results in multiple issues and (as mentioned in the previous paragraphs) requires additional time for testing and fixing errors. For example in Polish the string would read "*John karmi jego kot*", which is incorrect and even changes the meaning of the source. Figure 17.2 shows an example of concatenation in the GUI.

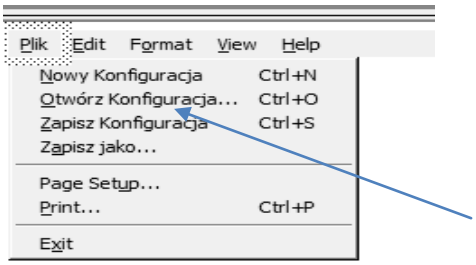

Fig. 17.2. Error caused by text concatenation

The errors mentioned above are just two popular examples that can be avoided if the development teams follow the internationalisation guidelines we list later in this chapter. There are many explanations given by developers why the rules were not applied – usually they reply that the end user group is limited (i.e. to only professionals), so the internationalisation errors seem less important. There is only one solution – developers need to follow *all* the internationalisation rules if the product is supposed to have multicultural support (http://www-01.ibm.com/software/globalization/guidelines/).

There are many cultural issues that are also included in the guidelines - they mainly apply to the author of the text and the localiser. The most important role of internationalisation is to allow smooth localisation and to create a proper technical environment for it. However, in some cases an incorrect cultural reference may be more serious to the end user than any technical error.

7. INTERNATIONALISATION EXAMPLES

Preparing a product for proper and more complex localisation, and therefore introducing internationalisation techniques, may be an extensive process that requires several versions of software. The gaming industry shows some interesting examples.

Figure 17.3 shows three development stages of the FIFA game series by Electronic Arts. The early versions only allowed subtitles. The next version also added Polish commentary. The newest version includes all the features introduced earlier, but goes even further - adding local leagues and players. Actually the packaging of the software is different in each country, which means almost each aspect of the product allows internationalisation. In this case the cover includes a local football player, Lewandowski.

Fig. 17.3. Several stages of internationalisation in FIFA game series by Electronic Arts. Source: Google

Fig. 17.4. Another example of game packaging internationalisation. Source: Google

It seems the producer (Electronic Arts) is also using this packaging internationalisation technique in other products. Figure 17.4 shows a cover of a military shooter game, which includes a well-known Polish special forces team called GROM. This is one of the many examples of how internationalisation brings the product closer to the end user. There is no doubt this encourages local customers to choose such a product.

One of the most important rules of internationalisation applies to text length. The example below shows a dialog box that would not be suitable in a case where the target language is significantly longer than the source. Internationalisation techniques help to properly redesign the dialog box.

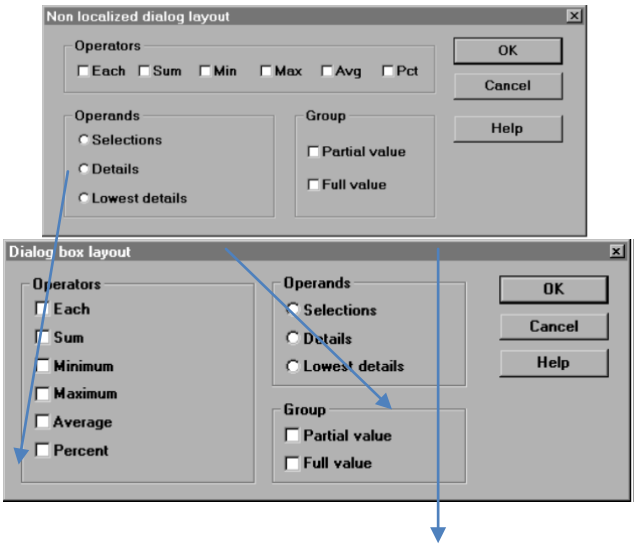

Fig. 17.5. Example of dialog box internationalisation. Based on lecture by Goldschmidt, & Zerfaß , 2010, Localization World 2010

8. INTERNATIONALISATION GUIDELINES

There are over 80 technical internationalisation rules that developers need to follow to create a product that can be localised into several languages. One of the most important general rules is "always separate text from code and move it to the project resources" (Sachse, 2012). This allows smooth localisation using software localisation tools. Hardcoded text (i.e. `MessageBox.("Sample text to be localised"`) in most cases cannot be extracted in a safe way. In the .NET environment this would require moving the text to .res/resx files, while in Java, text would need to be moved into .properties files (Property Resource Bundles). The biggest concern with strings embedded in the code is that all such issues will not be visible to the translator and can usually be detected only during the localisation testing stage, which is – as was said earlier – a very late and expensive way of fixing errors.

Another general rule is aimed at text authors/technical writers. To allow smooth translation resources should be created with an international audience in mind (http://www-01.ibm.com/software/globalization/guidelines/). Obviously the software needs to support the proper encoding (usually Unicode), fonts, correct direction and so on. Last but not least - all cultural assumptions need to be removed.

Please find below the complete list of technical rules specified in the Globalisation Design Guide by IBM (http://www-01.ibm.com/software/globalization/guidelines/):

- A1-1 Package MRI modules separately from the executable code.
- A1-2 Design applications so that they can be easily translated.
- A2 Consider Presentation Control Information (PCI) as part of the MRI and make the PCI available to the translator during the translation process.
- A2-1 Select data formats that are supported by your translation system.
- A3 Provide for effective presentation of MRI after expansion that results from translation.
- A4 Ensure that functions that depend on the location of user interface elements are not inhibited by display position changes caused by MRI expansion, or by mirroring of presentation for bidi languages.
- A5 Provide translators with a way of tracking MRI entities (such as messages and dialogs).
- A5-1 Have a mechanism to identify changes in the application or Web site and route that information into a translation process.
- A6 Permit variables that are substituted into text strings to assume any location and order.
- A7 Ensure that messages and other MRI are complete sentences; do not construct sentences from parts of sentences; do not construct words from word fragments.
- A8 Icons and clip art are to be treated as MRI and must, therefore, follow the rules for MRI (such as isolation, packaging and tracking) and be translatable.
- A8-1 Avoid text in icons.
- A8-2 Avoid humour, puns, slang, and special, mythological, and religious symbols in icons.
- A8-3 Avoid hand gestures or body positions in icons.
- A8-4 Allow for changes in icon colours because the meaning of a colour may differ between countries and regions.
- A8-5 Avoid implicit assumptions or expressions of reading and writing direction.
- A8-6 Suppress nonessential details in icons.
- A8-7 Avoid culturally specific metaphors in icons.
- A8-8 Use internationally recognized symbols in icons.
- A8-9 Avoid the use of national flags in icons.
- A9 Translatability of inputs such as commands, keywords, and responses is a must if the following conditions are met: they do not have a GUI equivalent, are not in common use in the industry and they can be found in the English dictionary.
- A10 Clearly identify and explain all trademark terms; verify all trademark claims and styles of acknowledgement with your intellectual property department.
- A11 When formatting text, be aware of punctuation marks or characters that cannot be placed at the beginning or end of a line during the line-breaking process.

- A12 Messages and data that go into customer log files must be translated while those that go into internal trace files must not be translated.
- A13 Mnemonics and accelerators are MRI and must therefore be translatable.
- A13-1 Consult and follow the IBM translation best practices when selecting symbols or letters for use in mnemonics or accelerators.
- A13-2 Do not use the AltGr key as the beginning of a mnemonic or accelerator
- B1 Write user interface information with an international audience in mind (for ease of translation into other languages, and for ease of understanding by non-English audiences).
- B1-1 Do not omit prepositions or articles from sentences.
- B1-2 Avoid noun strings.
- B1-3 Write in the active voice using the present tense.
- B1-4 Avoid using words in incorrect grammatical categories, such as using a noun or an adjective as a verb.
- B1-5 Avoid ambiguous pronoun references.
- B1-6 Use simple and clear coordination.
- B1-7 Ensure the elements of your sentences are parallel.
- B1-8 Filter out typographical errors.
- B1-9 Use correct terminology.
- B1-10 Be consistent with terminology.
- B1-11 Avoid referring to culture-specific standards.
- B1-12 Avoid using names that have meaning in your country or region only.
- B1-13 Avoid abbreviations, acronyms and special symbols: they cannot be translated.
- B1-14 Keep sentences as short and simple as possible.
- B1-15 Avoid the use of slang, jargon, humour, sarcasm, colloquialism, metaphor and informal everyday language.
- B1-16 Be concise.
- B1-17 Do not ask negatively phrased questions.
- B1-18 Do not use a slash to form the construction and/or; do not use a slash to mean and or or.
- B1-19 Avoid forming plurals by adding “(s)” to indicate either singular or plural form. Include both forms if necessary.
- B1-20 Refer to the symbol < ' > (apostrophe) as an apostrophe and the symbol < ’ > (single quote) as a single quotation mark.
- B1-21 Clearly identify non-cosmetic changes; avoid cosmetic changes that affect translation already done.
- C1 Allow the Selection of Calendar and Calendar Format
- C2 Allow the Selection of Date and Time Format
- C3 Allow the Selection of Time zone
- C4 Allow the Selection of Paper Size
- C5 Allow the Selection of Cardinal Number Shape

- C6 Allow the Selection of Numeric Value Format
- C7 Allow the Selection of Monetary Amount Format
- C8 Allow the Selection of Mathematical Format
- C9 Allow the Selection of Measurement System
- C10 Allow the Selection of Sentence Spacing and Punctuation
- C11 Allow the Selection of First Day of Week
- C12 Allow the Customization of Address Format
- C13 Allow the Customization of Telephone Number Format
- C14 Allow the Customization of Name Format
- C15 Ensure conformance with Government Regulations
- C16 Allow the Selection of User-Interface Language
- D1 – Isolating Culture-sensitive and Language-sensitive Parts
- D2 – Ensuring the Compatibility of Language and Culture Parts
- D3 – Structuring Services for Full Support
- D4 – Providing for Language Exits
- D5 – Selecting Development Tools
- E1 – Counting Graphic Keys
- E2 – Controlling The Operating Modes
- E3 – Switching Graphic Groups
- E4 – Supporting Capital Lock or Level 2 Lock Function
- E5 – Implementing Capital Lock Function
- E6 – Defining a Combining or Repeating Key
- E7 – Positioning of Graphics
- E8 – Providing for Flexible Character Generator
- E9 – Providing Consistent Presentation Intensity
- E10 – Compensating for Printer Duty Cycle Limitations
- E11 – Accommodating Different Paper Sizes
- F1 – Coding Graphic Characters
- F2 – Using Graphic Characters
- F3 – Supporting Graphic Character Sets
- F4 – Accessing Graphic Characters
- F5 – Validating Graphic Characters
- F6 – Respecting Reserved Code Points
- F7 – Redefining Graphic Character Meaning
- F8 – Avoiding Unassigned Code Points
- F9 – Identify Encoding
- G1 – Allowing Coexistence of MBCS and SBCS Data
- G2 – Recognizing Multibyte Characters
- G3 – Manipulating MBCS Data Stream
- G4 – Converting Multibyte Characters
- G5 – Buffer Space Considerations
- G6 – Switching Character Interpretation

- G7 – Adding New Characters
- H1 – Entering Bidirectional Text
- H2 – Respecting the Bidirectional Attributes
- H3 – Mirroring of Graphic User Interface Elements
- H4 – Isolating Artwork Orientation and Text Orientation

All large software manufacturers have their own internationalisation guides, so it is strongly recommended to select the one most relevant to the development framework used in the specific project.

The most popular error detected during testing is truncated text. One of the reasons for this is not applying internationalisation rules related to the length of text. For many language pairs we often see text expansion, meaning that the target string (or the content of text boxes) can be up to 20-30% longer (Zerfaß, 2010) once the translation is completed. In rare language variants we may see text contraction, but in general we should always expect that our source text may need to be significantly longer in the target language version. Fortunately many programming frameworks now support autoresizing of buttons and dialog boxes. Figure 17.6 shows and example of truncated strings:

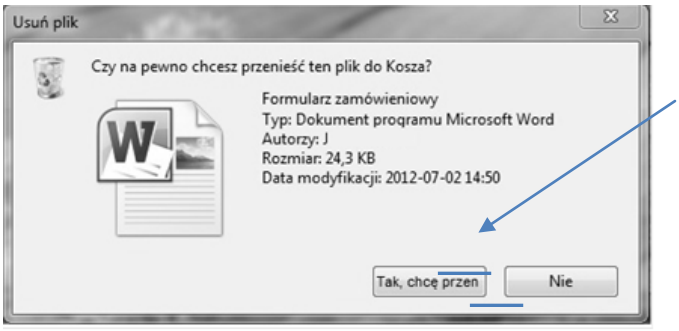

Fig. 17.6. Example of truncated text

Also one of the often reported errors is related to date, calendar and measurements format that cannot be changed to the proper local appearance. Figure 17.7 shows an example of incorrect calendar format for Polish.

Correct form: 20 lipca 2012 or 2012-07-20

Fig. 17.7. Example of calendar issues

Other common error is also enforcing a calendar with Monday being the first day of the week. In general - not following any of the rules mentioned in the list above leads to errors in translated products.

It is worth mentioning that most of the described techniques can also be used in products other than software. For example a car manufacturer would use internationalisation techniques to allow assembling the driving wheel on both sides and localising manuals to several languages. This applies to many domains, i.e. power connectors are the most commonly internationalised hardware element. All social media portals are also "i18n-ready" by design.

Please note this paper does not refer directly to the very important issue of the source text quality. There are many non-technical issues, which may occur when the quality of the source document is low, usually when it is not written by native speakers.

9. CONCLUSION

Internationalisation techniques allow a significant decrease of the error rate in software projects and smooth product localisation: crucial elements in reaching multilingual customers worldwide. Many examples show the consequences of incorrect product design and the impact on localisation - and therefore on product sales on the specific market.

Yet, convincing companies to focus on internationalisation before and during the development phase is still work in progress. Once the decision to introduce a full internationalisation process has been made, it is important to refer to a specific internationalisation guide with all rules explained in details. Please note only a small number of rules and examples were explained in this chapter.

REFERENCES

[1] Bartnicka M. & Gacka R., (2012), "Software internationalization in IBM" (in Polish), Lecture at Silesian Univeristy.
[2] Deitsch A. & Czarnecki D., (2001), "JAVA Internationalization", O'Reilly.
[3] Esselink B., (2000), "A Practical Guide for Localization" (2nd Edition), John Benjamin Pub. Co.
[4] Eve P., (2012), "Multilingual e-commerce: US and UK trends", http://econsultancy.com/uk/blog/10130-multilingual-e-commerce-us-and-uk-trends.
[5] Goldschmidt D., Zerfaß A., 2010, "Best Practices Areas in Internationalization/Localization", Localization World 2010 Conference Materials.
[6] Hall B., (2007), "Globalization Handbook For The Microsoft .NET Platfrom", Multilingual Press.
[7] Kahn K. B., (2004), "The PDMA Handbook of New Product Development", Second Edition, John Wiley & Sons, 2004. Chapter 12, pp. 173-187.

[8] Kano N, (2002),"Developing International Software by Dr. International", Microsoft Press.
[9] Kaplan M. S., (2000), "Internationalization with Visual Basic", Sams.
[10] Kelly J. C., Sherif, J.S. & Hops J., (1992), "An analysis of defect densities found during software inspections",Journal of Systems and Software archive, Volume 17 Issue 2, Pages 111 - 117, Elsevier Science Inc. New York, NY, USA
[11] Kelly N., DePalma D. A. & Stewart R. G., (2012),"The Language Services Market", page 2.
[12] IBM, (2011), "Globalisation Design Guide, http://www-01.ibm.com/software/globalization/guidelines/
[13] IBM, (2011), "Multicultural Support Reference Guide", http://pic.dhe.ibm.com/infocenter/aix/v7r1/topic/com.ibm.aix.nls/doc/nlsgdrf/nlsgdrf_pdf.pdf.
[14] IBM, (2012), "Bidirectional Support Guide", http://publib.boulder.ibm.com/infocenter/hatshelp/v75/index.jsp?topic=/com.ibm.hats.doc/doc/ugbidi.htm.
[15] Savourel Y., (2001), "XML Internationalization and Localization", Sams, Indianapolis.
[16] Sachse F., (2012), "How to deliver software for global audiences: An introduction to software internationalization", Webinar recording by SDL.
[17] Symmonds N., (2002), "Internationalization and Localization Using Microsoft .NET", Apress.
[18] Vadher P., (2010), http://www.softwaretestingdiary.com/2010/08/software-testing-and-quality-assurance.html.
[19] Zydroń A., (2004), "Coping with Babel: How to Localize XML", The Globalization Insider, Volume XIII, Issue 4.2.

The author would like thank Ms. Marta Bartnicka for her help in obtaining the valuable data used in this publication and sharing her comments.